1 Format

The exam will be 3 hours long. It may have true/false questions, multiple choice, example / counterexample problems, run-this-algorithm problems, and problem set style present-and-prove problems.

As usual, whenever you are asked to give an algorithm for a problem, you are expected to analyze its correctness and running time. More efficient correct solutions receive more credit. Less efficient, but correct solutions will receive partial credit unless otherwise stated. We understand and expect that proofs of correctness will be more concise on exams than they would be on homework.

The final will focus on topics covered after Quiz 2 (randomized algorithms and beyond), but questions from any part of the course. The topics we have covered include, but are probably not limited to those in the following topics list.

2 Quiz 1 Topics

2.1 Math/Fundamentals

- Induction. If \( P(n) \) is a statement (“\( 2n \) is even”), \( P(1) \) is true, and \( \forall n, P(n) \rightarrow P(n + 1) \), then \( \forall n, P(n) \) is true.

- Big-O Notation. \( o, O, \omega, \Omega, \Theta \) and identifying which of these relations hold for two functions.

- Recurrence Relations. Solve simple ones by finding a pattern and proving them with induction. More complicated recurrences can be solved using the Master Theorem (must memorize).

- Integer Multiplication. Perform 3 multiplications on \( n/2 \) digit numbers, and then do some additions.

- Fast Powering. Use repeated squaring to find an \( n \)th power in \( O(\log n) \) time.

- Merge Sort. Sort a list of \( n \) numbers in \( O(n \log n) \) time. Implement recursively or iteratively.

2.2 Graph Search

- Representation. Adjacency list versus adjacency matrix.

- Depth First Search (DFS). Uses a stack to process nodes, Pre and post order numbers, labels for the edges (tree, forward, back, cross).

- Breadth First Search (BFS). Uses a queue to process nodes, can be used to find the shortest path when all edge weights are 1.
• **Dijkstra’s Algorithm.** Single source shortest path for non-negative edge weights. Uses a heap or priority queue to process nodes. Does not work when there are negative edge weights (why?).

• **Heaps.** Binary heap implementation, operations: `DELETEMIN`, `INSERT`, `DECREASEKEY`, how they are used in Dijkstra’s algorithm.

• **Bellman-Ford Algorithm.** Single source shortest path for general edge weights, edge relaxation procedure (referred to as `update` in the lecture notes), detecting negative cycles.

• **Floyd-Warshall Algorithm.** All pairs shortest paths via dynamic programming, detecting negative cycles.

• **Shortest Path in DAG.** Can be done in linear time via dynamic programming regardless of edge weights.

3 Quiz 2 Topics

3.1 Minimal Spanning Trees

• **Prim’s Algorithm.** Uses a min heap. There are \( n \) `INSERT` operations, at most \( n \) `DELETEMIN` operations, and at most \( m \) `DECREASEKEY` operations. Therefore, using a binary heap, the runtime is \( O((m+n) \log n) = O(m \log n) \). With a Fibonacci heap, the runtime is \( O(m+n \log n) \).

• **Kruskal’s Algorithm.** Uses the Union-Find data structure (see next subsection). Takes \( O(m \log m) \) time to sort, but then takes \( O(m \log^* m) \) time for the union-find part.

3.2 Union Find

• **Main Idea.** Disjoint set data structure that supports the operations `UNION` and `FIND`.

• **Union by Rank and Path Compression.** You are able to achieve \( O(m \log^* n) \) with both Union by Rank and Path Compression.

3.3 Greedy

• **Main Idea.** At each step, make a locally optimal choice in hope of reaching the globally optimal solution.

• **Biohazard!** Remember that greedy algorithms can often seem correct, so it’s extra important to prove the correctness and optimality of your algorithm.

• **Horn Formulae.** Set all variables to false, and greedily set variables as true when forced to.

• **Huffman Coding.** Find the best encoding by greedily combining the two symbols of lowest frequency.

• **Set Cover.** We have a greedy approximation algorithm with \( O(\log n) \) performance ratio.
3.4 Divide and Conquer

- **Main Idea.** Divide the problem into smaller pieces, recursively solve those, then combine them in the right way.
- **Mergesort.**
- **Integer Multiplication.** Perform 3 multiplications on \( n/2 \) digit numbers, and then do some additions.
- **Strassen’s Algorithm.** Multiplies two \( n \times n \) matrices by doing 7 multiplications on \( n/2 \times n/2 \) matrices.

3.5 Fast Fourier Transform

- **Main Idea.** We can multiply to polynomials of degree \( n \) in \( O(n \log n) \) time.
- **Integer Multiplication.** We can treat an (base-10) integer as a polynomial evaluated at 10.
- **Successive Dot Product.** By cleverly selecting the coefficients on our polynomials (reversing the coefficients on the \( j \)-degree polynomial), for polynomials of degree \( j \) and \( k \), with \( j \leq k \), we can compute the \( k - j + 1 \) successive dot products (shifting \( j \) coefficients along the \( k \) coefficients) simply by multiplying these polynomials together in \( O(k \log k) \) time.
- **Pattern Matching.** To search for the existence of a pattern \( P \) of length \( m \) in a string \( T \) of length \( n \geq m \), we can use SDP. We saw in pset 5 problem 3 that we can get \( O(n \log m) \) time for arbitrary sized alphabets with wildcard symbols.

3.6 Dynamic Programming

- **Main Idea.** Maintain a lookup table of correct solutions to sub-problems and build up this table in a particular order.
- **All Pairs Shortest Paths.** Uses the idea that the shortest path to a node must have come via one of the neighboring nodes.
- **String Reconstruction.** Tries to find where the first dictionary word ends by checking all possible breaking points.
- **Edit Distance.** Tries all possibilities of INSERT, DELETE, CHANGE on letters that are different.
- **Matrix Chain Multiplication.** We take advantage of the associativity of matrix multiplication to find the optimal parenthesization to minimize the cost of multiplying a chain of matrices (of varying dimensions) together.
- **Traveling Salesman.** DP can provide better exponential-time solutions to NP hard problems.

You should treat dynamic programming problems as having three parts:

(a) **Define \( f \) in words** (without mention of how to compute it recursively). You should clearly state how many parameters \( f \) has, what those parameters represent, what \( f \) evaluated on those parameters represents, and what parameters you should feed into \( f \) to get the answer to the stated problem.
(b) Define $f$ in math. Give a recurrence relation showing how to compute $f$ recursively.

(c) Give the running time and space for solving the original problem using computation of $f$ via memoization or bottom-up dynamic programming. If you need to use certain data structures to make computation of $f$ faster, you should say so.

If there are multiple solutions to solve the stated dynamic programming problem, you should describe the most time efficient one you know. If there are multiple solutions with the same asymptotic time complexity, you should describe the implementation that gives the best asymptotic space complexity.

4 Post Quiz 2 Topics

4.1 Randomized Algorithms

- **Main Idea.** A randomized algorithm rely on random decision-making (“coin tosses”). There are two main types, Las Vegas with certain correctness and random runtime, and Monte Carlo with random correctness and fixed runtime.

- **Frievalds’ algorithm.** We select $k$ random binary vectors $x_1, \ldots, x_k \in \{0, 1\}$ and if $Cx_i = ABx_i$ for all $i = 1, \ldots, k$, then we output $C = AB$. Otherwise, we output $C \neq AB$. Monte Carlo with runtime $\Theta(kn^2)$ and probability of failure $1/2^k$.

- **QuickSort.** We select a pivot element, and place the pivot element in its correct spot. Then we recursively sort the left and right subarrays. Las Vegas with expected runtime of $\Theta(n \log n)$.

- **QuickSelect.** We want to find the $k^{th}$ smallest element. We randomly select a pivot element, and recurse on the subarray containing the $k^{th}$ smallest element. Las Vegas with expected runtime of $O(n)$ time.

4.2 Randomized Data Structures

- **Skip Lists.** A skip list supports INSERT($k$), DELETE($x$), and PRED($k$). It maintains all $n$ items in a linked list at the bottom level (level 0 for this description) in sorted order. We flip a fair coin to determine whether an element in level $i$ also lives in level $i + 1$ for $i = 0, 1, 2, \ldots$. We expect half of the elements from a level to advance to the next level, so the expected space is $O(n)$, with expected time to query any item $i \in \{1, \ldots, n\}$ to be $O(\log n)$.

- **Dictionary Problem.** We have (key, value) pairs with keys in the range $\{1, \ldots, U\}$. In the dynamic dictionary problem, we have the operations INSERT($k$, $v$), DELETE($x$), and QUERY($k$). In the static dictionary problem, we support QUERY($k$) only.

- **Hashing.** A hash family is a set of functions $h : \{1, \ldots, U\} \to \{1, \ldots, m\}$. We seek to have relatively simple hash functions that have low probability of collision, which is when two distinct keys $k, k'$ for which $h(k) = h(k')$. Some extreme examples of hash functions are $h(i) = 0$ for all $i$ (many collisions), and $h(i) = i$ for all $i$ (no collisions).

- **Universal Hash Family.** A family $\mathcal{H}$ of hash functions from $\{1, \ldots, U\} \to \{1, \ldots, m\}$ is universal if for all $1 \leq x < y \leq U$, and for all $h \in \mathcal{H}$ that $P(h(x) = h(y)) \leq 1/m$. 

4
4.3 Network Flow

- **Main Idea.** We consider our graph as a flow network, and we send an amount of flow from a source \( s \) to a sink \( t \) considering the capacity constraint (our pipes can’t hold infinite water) and flow conservation (water cannot disappear).

- **Ford-Fulkerson Algorithm.** Used to find the maximum flow in a flow network. Keeps track of the residual graph, and uses DFS to find paths to send water from \( s \) to \( t \) in the residual graph. Has runtime \( O((n + m)f^*) \) where \( f^* \) is the max flow. When we use BFS instead, we have the Edmonds-Karp algorithm, which has a runtime of \( O(nm^2) \) which does not depend on \( f^* \).

- **Bipartite Matching.** We can solve the bipartite matching problem by adding a source \( s \) connected to all vertices of the first color, adding a sink \( t \) connected to all vertices of the second color, then running Ford-Fulkerson, or otherwise finding the max flow. For example, matching frogs to consumable flies in Problem Set 8 where we were looking for a perfect matching (all vertices are matched up with no shared edges) between frogs and flies.

- **Min-Cut Max-Flow Theorem.** The minimum cut required to disconnect a graph with \( s \) and \( t \) on opposite sides of the cut is equal to the maximum flow we can send from \( s \) to \( t \).

4.4 Linear Programming

- **Main Idea.** We want to minimize or maximize some objective function of the form \( a_1x_1 + \cdots + a_nx_n \) with respect to a set of \( i \) constraints \( c_{i,1}x_1 + \cdots + c_{i,k}x_k \geq b_i \). The coefficients on our variables \( x_1, \ldots, x_n \) take on the real numbers and can be zero. We can solve such a system using Simplex (which we treat in this course as a black box).

- **Duality.** The concept of duality generalizes the idea of the min-cut max-flow theorem to any linear program.

- **Zero Sum Games.** To use linear programming to analyze zero sum games, we define variables \( z \) the expected winnings for the Row player, and \( w \) the expected losses for the Column player. We can write constraints to maximize \( z \), and we can also write constraints to minimize \( w \). The optimal value for the objective function for either of these linear programs is the same, and we call this value the “value of the game”.

4.5 NP Completeness

- **Definitions.** The class \( P \) is the set of all problems with yes/no answers that can be answered in polynomial time. That is, the solution can be found in \( O(n^k) \) steps where \( k \) is some non-negative integer. The class \( NP \) is the set of all problems with yes/no answers that can be verified in polynomial time using a short certificate. That is, when the answer is yes, then given the input and the short certificate, we can verify the answer as yes in \( O(n^k) \) time.

- **Reductions.** A reduction is a procedure that transforms an input for a problem into an input for another problem. If we reduce a problem \( A \) to another problem \( B \), then give or take a polynomial, \( A \) can be no harder than \( B \) (we at least have \( B \) as a way to solve \( A \)).

- **NP-Completeness.** NP-complete problems are the “hardest” problems in \( NP \), and all other problems in \( NP \) reduce to them. To show that a problem is NP-complete, we reduce a known
**NP**-complete problem to it. Known **NP**-complete problems include Circuit-SAT, 3-SAT, Integer Linear Programming, Independent Set, and Vertex Cover.

- **Approximation Algorithms.** An approximation algorithm gives a solution to a problem that is not optimal. We can describe how suboptimal an approximation algorithm is by computing the approximation ratio. Approximation algorithms can serve as polynomial time approximations for **NP**-complete problems, such as Vertex Cover, which we can approximate with an approximation ratio of 2.